Near Real-Time CSG Rendering using Tree Normalization and Geometric Pruning

TR88-006

1988

N00014-86-K-0680

Jack Goldfeather, Steven Molnar, Greg Turk, and Henry Fuchs

The University of North Carolina at Chapel Hill
Department of Computer Science
Sitterson Hall, 083A
Chapel Hill, NC 27599-3175

Near Real-Time CSG Rendering using Tree Normalization and Geometric Pruning

Jack Goldfeather*, Steven Molnar, Greg Turk and Henry Fuchs

University of North Carolina
Chapel Hill, NC 27514

11 January 1988

Abstract

Described is a set of algorithms for efficiently rendering a CSG-defined object directly into a frame buffer without converting first to a boundary representation. The method requires only that the frame buffer contain sufficient memory to hold two color values, two z values, and three one-bit flags. The algorithm first converts the CSG tree to a normalized form that is analogous to the sum-of-products form for boolean switching functions. Expanding on results previously ~~first~~ reported ~~in [GHF86], we~~ the authors introduce ~~here~~ 1) an improved algorithm for converting a CSG expression to sum-of-products form, 2) a technique for minimizing the tree using "don't care" conditions based on geometry, 3) the dynamic interleaving of the above two algorithms, allowing very efficient rendering for most CSG objects, 4) a method for extending the technique to non-convex primitives, and 5) an implementation of these ideas in an interactive CSG design system on Pixel-planes 4. In this system the designer directly manipulates the CSG structure while continuously viewing the color rendering of the object under design. It is believed ~~We believe~~ that these ~~our~~ algorithms will attain similar speeds on many of the next generation high-performance graphics systems that have frame buffers with many bits per pixel. (KP)

*Jack Goldfeather is professor of mathematics at Carleton College, Northfield, Minnesota

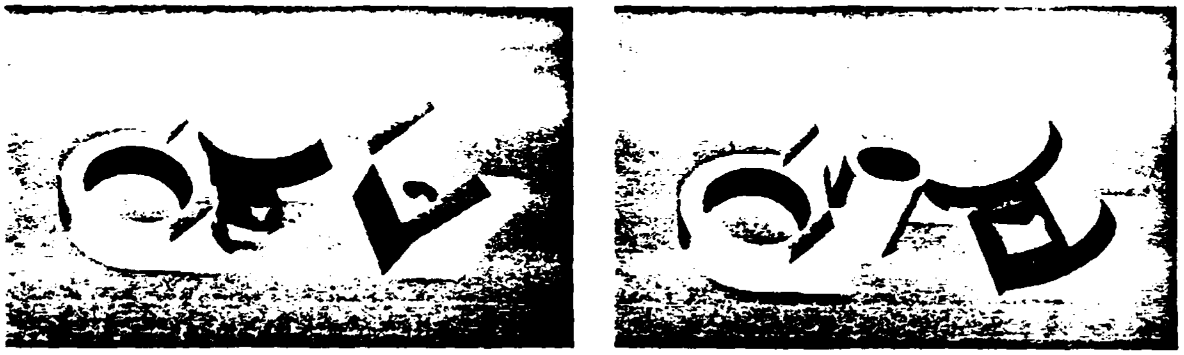

Figure 1: Consecutive frames during an editing session (0.142 seconds per frame)

1 Introduction

A serious bottleneck in using Constructive Solid Geometry (CSG) to design 3D objects is the time required to generate shaded images [RR86][For86]. Although at least one team has proposed and is now building specialized hardware for rapid rendering of CSG-defined objects [KE84], we know of no system today, other than the one we present in this paper, that can render non-trivial CSG-defined objects in near real-time.

1.1 Background

Three general methods have been proposed for fast CSG displays: 1) conversion to boundary-representation, 2) scan-line algorithms, and 3) depth-buffering algorithms.

Conversion to boundary representation (BREP) is the traditional method used in many CSG modeling systems designed when direct rendering from the CSG description was very time-consuming. It allows rapid viewing of the object using conventional rendering engines, but requires a BREP conversion every time the object is modified, and is therefore infeasible for systems designed to display an object interactively during editing sessions. The University of Rochester's PADL-2 [Bro82], Brown University's BAGS [Cha86], General Motor's GMSolid [BG82], and Ricoh's DesignBase [TSUC86] systems all use BREP conversion to generate shaded images.

Atherton published a CSG scan-line algorithm in [Ath83]. Scan-line algorithms display directly from the CSG description, but are slow when implemented on a conventional machine. Scan-line algorithms can be sped up using object-parallelism. Kedem and Ellis currently are building a parallel ray-casting machine for fast scan-line rendering [KE84]. Their machine allocates one processor per node in the CSG tree. An arbitrary CSG tree with number of nodes less than the number of hardware processors can be rendered directly; larger trees must be rendered using multiple passes.

Depth-buffering algorithms involve generalizations of the z-buffer hidden surface algorithm. Speed can be gained using pixel-parallelism in special-purpose hardware. Rossignac and Requicha [RR86] published an algorithm for directly rendering CSG objects using point-classification and z-buffering. Jansen [Jan86] published a similar algorithm specifically designed for pixel-parallel machines like Pixel-planes. Both the rendering algorithms of Rossignac/Requicha and Jansen are based on the original CSG tree configuration. A number of flags ($\log_2 n$ bits, where n is the number of primitives) are needed to hold intermediate point-classification results while traversing the tree. Our approach, first presented in [GHF86], rearranges the CSG tree before rendering, eliminating the need for these "place-holding" bits. Okino, Kakazu, and Morimoto [OKM84] published a paper on depth-buffering algorithms which also may employ normalization, but we have been unable to verify this.

Naylor [TN87] demonstrated dynamically rendered images with a recent system using Binary-Space Partitioning Trees, but their system can handle only a limited set of modifications without incurring many seconds of delay. For example, moving a beveled hole would be time-consuming.

The algorithm we present here will render any CSG object using a constant number of bits per pixel (≤ 128), in many cases more efficiently than the Requicha/Jansen approach. Our rendering system, based on this algorithm, displays modestly-sized CSG objects in fractions of a second, even if the user changes the geometric structure of the CSG tree each and every frame.

1.2 Overview of Paper

Section 2 details the tree-normalization/rendering process first presented in [GHF86]. We will take the opportunity to correct a small error in the normalization algorithm described there. In this section we will restrict the class of CSG objects that can be rendered to objects with convex primitives. Later (Section 4) we will loosen this restriction. Section 3 introduces geometric pruning, an application of bounding boxes to eliminate unnecessary subtrees. We will observe the useful phenomenon that, although normalization tends to increase the number of nodes in the CSG tree, pruning not only cuts the size of the tree considerably, but works most effectively during the process of normalization. Section 4 describes an extension of the the rendering algorithm to handle non-convex primitives. Section 5 describes our interactive modeling system running on Pixel-planes 4 and gives statistics for the normalization/geometric pruning (NGP) algorithm applied to various CSG objects.

2 CSG Rendering using the Normalization Method

Normalizing a CSG tree is analogous to converting a boolean expression to sum-of-products form. A boolean expression written as a sum of products has a two-level AND/OR realization. This enables fast combinational logic with only two gate delays (for example, see [Koh78, page 84]). A CSG expression written as a sum of products can be rendered using two *image* (z and color) buffer pairs. This enables any CSG object composed of convex primitives to be rendered in a frame buffer with a constant number (2∗length(z buffer) + 2∗length(color buffer) + 3) bits per pixel. Figure 2 diagrams the relationship between a two-level realization in combinational logic and the NGP method of rendering CSG images.

2.1 Boolean Representation of a CSG Tree

Any CSG tree can be represented as a boolean expression by replacing each union ($\cup$) by a sum, each intersection ($\cap$) by a product, and each difference ($-$) by a product of a complement. The

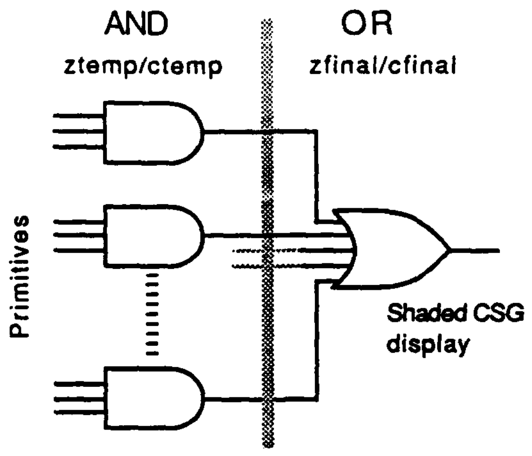

Figure 2: Boolean Logic/CSG Rendering Analogy

primitives in a CSG tree correspond to *literals* in the boolean expression. For example, the CSG tree $((A-B)\cup C)\cap D$ becomes $(AB'+C)D$. We will use whichever notation is more convenient. In particular, we will define a *normalized* CSG tree in terms of well known boolean constructs.

Definition: A CSG tree is in *normal* form if its boolean representation is in disjunctive normal form, i.e. if it is a sum of products of literals.

In addition to allowing rendering with a constant number of bits per pixel, normalizing a CSG tree allows the rendering algorithm to be simpler than it would be otherwise. Each product in the normalized expression can be rendered using primitive/primitive interaction rather than subtree/subtree interaction. In Section 3 we will see that normalization also allows unnecessary portions of the CSG tree to be recognized and pruned very easily.

2.2 The Normalization Algorithm

The normalization algorithm uses the eight basic set equivalences of Figure 3 to reduce a CSG tree to normal form. These equivalences encapsulate the associative and distributive properties of set operations and were chosen because they represent all of the possible unnormalized

1. $X - (Y \cup Z) = (X - Y) - Z$
2. $X \cap (Y \cup Z) = (X \cap Y) \cup (X \cap Z)$
3. $X - (Y \cap Z) = (X - Y) \cup (X - Z)$
4. $X \cap (Y \cap Z) = (X \cap Y) \cap Z$
5. $X - (Y - Z) = (X - Y) \cup (X \cap Z)$
6. $X \cap (Y - Z) = (X \cap Y) - Z$
7. $(X \cup Y) - Z = (X - Z) \cup (Y - Z)$
8. $(X \cup Y) \cap Z = (X \cap Z) \cup (Y \cap Z)$

Figure 3: Set Equivalences for Normalization

configurations at a single node. Equivalences 2,3,5,7, and 8 decompose an expression into sums, and the remaining equivalences replace right association with left association. The normalization algorithm can be written recursively as follows:

```
normalize(T:tree):tree;
    { reduce a CSG tree to sum-of-products form }
    begin
        if (T=PRIMITIVE) then return(T);
        else
            begin
                while ((T $\neq \cup$) and ((T.right $\neq$ PRIMITIVE) or (T.left = $\cup$))) do
                    begin
                        if (T $\neq \cup$) then apply appropriate set equivalence (1-8)
                            1-6 before 7 or 8;
                        normalize(T.left);
                    end;
                normalize(T.right);
                return(T);
            end;
    end normalize;
```

This version corrects an error in the algorithm given in [GHF86, page 113]. Figure 4 shows a sample CSG tree before and after normalization. The algorithm above is not the only possible normalization algorithm, since there are many ways to convert a boolean expression to disjunctive normal form. We have proven, however, that *normalize* has the following desirable properties:

1. It terminates given any CSG tree as input.
2. If it terminates, it will leave the CSG in normal form.
3. Each restructuring step requires only local information (node type and child node types).
4. If the initial tree contains no redundant subtrees or repeated primitives, it will not add redundant product terms or repeated primitives within a product.

Property 3 follows directly from the set equivalences. We have developed formal proofs for properties 1,2 and 4 [GM88], but have not included them here for lack of space.

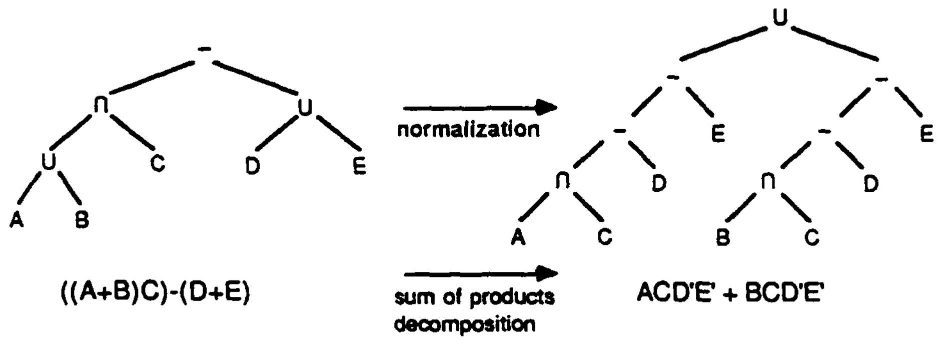

Figure 4: CSG tree before and after normalization

2.3 Displaying a CSG Tree in Normal Form

In this section we describe how a CSG tree in normal form can be rendered using only two z/color buffer pairs: $(ztemp, ctemp)$ and $(zfinal, cfinal)$. We will describe the process for convex primitive solids now and refer the reader to Section 4 for an extension of the rendering algorithm to non-convex solids. The basic idea is to break each product into separate *terms*, render each term into the $(ztemp, ctemp)$ image buffer, then composite the terms into the $(zfinal, cfinal)$ image buffer using a standard z-buffer algorithm. A product with n primitives will contain n terms. Rendering each term consists of rendering the surface of a primitive and paring it by the remaining $n-1$ primitives in the product.

In the following algorithm a primitive is *complemented* if the corresponding literal in the normalized boolean expression is complemented. $Zfar$ is a constant equal to the largest value representable in either z-buffer. The front surface of a primitive refers to all of the points on the exterior of a primitive that are visible from the eye point. In our case this is simply the set of front-facing polygons. The algorithm for rendering a product is:

```
procedure front(P:primitive);
    { place front surface of P into ($ztemp, ctemp$) }

procedure back(P:primitive);
    { place back surface of P into ($ztemp, ctemp$) }

procedure intersect(P:primitive);
    { if $ztemp$ not between front and back surfaces of P, set $ztemp = zfar$ }

procedure subtract(P:primitive);
    { if $ztemp$ between front and back surfaces of P, set $ztemp = zfar$ }

procedure render(M:product);
    begin
        for each primitive P in M do
            begin
                if P is uncomplemented then front(P) else back(P);
                for each primitive Q ($Q \neq P$) in M
                    if Q is uncomplemented then intersect(Q) else subtract(Q);
            end;
    end render;
```

Figure 5 illustrates the entire rendering process for the CSG tree $((A - B) \cup C) \cap D$. The photographs illustrate the contents of the temporary and final image buffers at each stage in the rendering process.

3 Geometric Pruning

Normalization can add many primitive leaf nodes to a CSG tree; for certain trees the increase can be dramatic (sample object 11 in Figure 8 expanded from 42 primitives before normalization to 69120 primitives after normalization). In many such cases major subtrees in the normalized tree will not contribute to the final image since primitives within them may not intersect. Information about which primitives overlap provides us with "don't care" conditions similar to those used

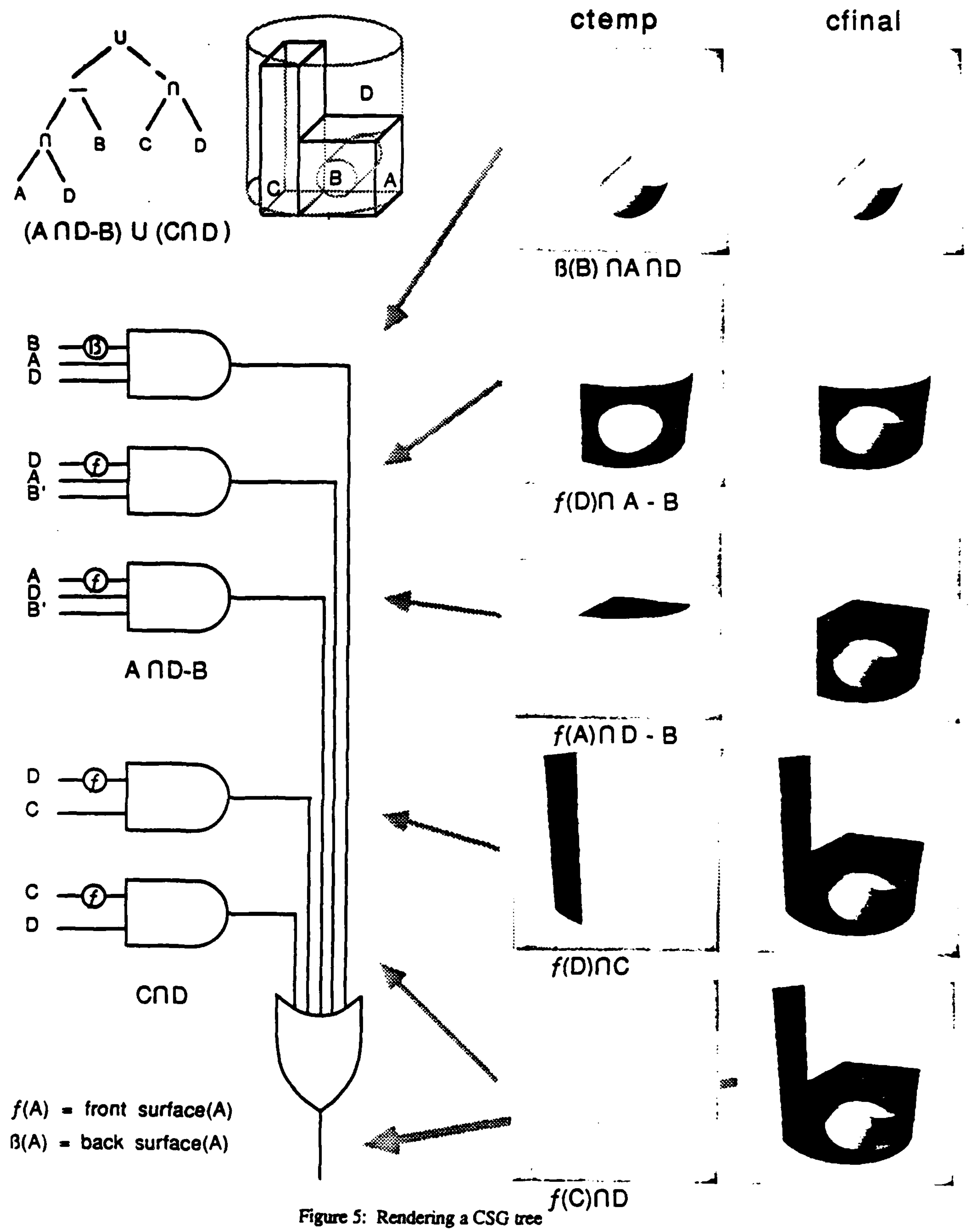

Figure 5: Rendering a CSG tree

to minimize logic expressions. We will describe how to prune normalized trees using geometric extent (bounding box) information derived from the geometry of the objects. We will see that this type of pruning is much more effective than pruning done without normalization.

3.1 Bounding Boxes

The idea behind geometric pruning is to calculate a few numbers that bound the region occupied by a CSG object. Any bounding region can be used; we choose to use bounding boxes because of their simplicity. Only if the bounding boxes intersect do we need to compute the actual intersection between objects. Similarly for subtraction, if the bounding box of an object to be subtracted does not intersect the bounding box of the object from which it will be subtracted, the subtraction does not need to be computed. Entire subtrees can be eliminated as well as primitives, since each subtree has a bounding box derived from the subtrees below it. Other authors have noted this as well [JS87][BMPA87].

3.2 Using Bounding Boxes to Prune a CSG Tree

To compute bounding boxes for primitive nodes, we must first transform them into a common coordinate system; eye space is a natural choice. We then compute a bounding box for each primitive by finding the minimum and maximum x, y, and z coordinates of the vertex set. $Min(x, y, z)$ and $Max(x, y, z)$ define the corners of a box aligned with the coordinate axes that completely encloses the primitive.

We compute bounding boxes for an operator node from the bounding boxes of its children using the following rules:

1. Bounding Box(A $\cup$ B) = Bounding Box(A) $\cup$ Bounding Box(B)
2. Bounding Box(A $\cap$ B) = Bounding Box(A) $\cap$ Bounding Box(B)
3. Bounding Box(A $-$ B) = Bounding Box(A)

Intersection and difference nodes can potentially be pruned. If the operator is $\cap$, we compare $Min(x, y, z)$ with $Max(x, y, z)$. If the minimum in any coordinate exceeds the maximum, the

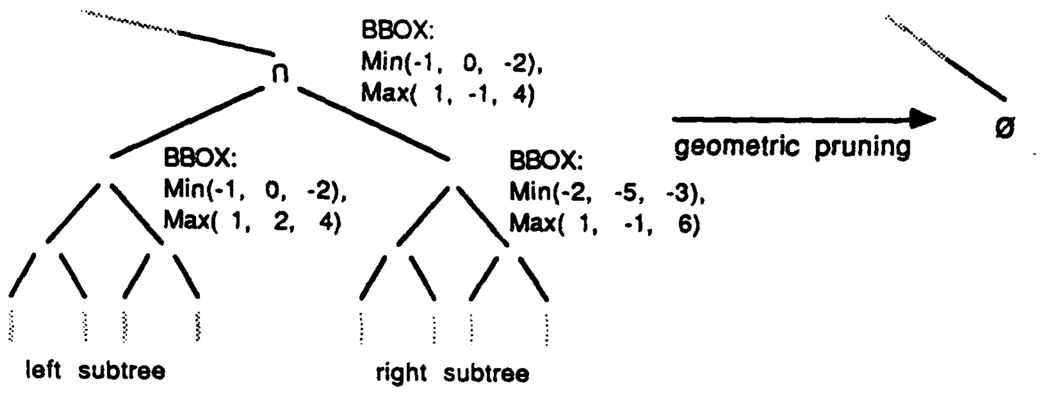
∩
BBOX: Min(-1, 0, -2), Max(1, -1, 4)
BBOX: Min(-1, 0, -2), Max(1, 2, 4)
BBOX: Min(-2, -5, -3), Max(1, -1, 6)
geometric pruning
Ø
left subtree
right subtree

Figure 6: Pruning an intersection node

bounding box describes a null volume and we prune the subtree A ∩ B. If the operator is −, we compare Bounding Box(A) with Bounding Box(B). If they do not intersect, we replace the subtree A − B with A. We can prune subtrees that do not intersect the viewing frustum as well. Figure 6 illustrates pruning an intersection node.

3.3 Places to Prune

There are several places in the display process where we can prune a CSG tree:

- before normalization (with primitives transformed to eye space).
- after normalization.
- *during* normalization.

Pruning after and during normalization will produce the same results; pruning before normalization will not. Each location has advantages and disadvantages. While an unnormalized tree, in general, contains fewer nodes than a normalized tree, the CSG tree designer is not likely to include nodes that do not contribute to the final image. We have found pruning at this stage to be ineffective on all of the CSG data sets we have rendered (see Figure 8).

Normalization, even though tending to increase the number of tree nodes, separates large subtrees (with large bounding boxes) into a set of product terms (generally with smaller bounding boxes). This makes pruning very effective on normalized trees, in some cases reducing the number of primitive leaf nodes by two orders of magnitude. Geometric pruning reduced the number of primitives from 69120 to 96 in sample object 11 of Figure 8.

An even more effective approach is to prune the tree before it ever reaches its maximum size. We can do this by pruning *during* normalization rather than after. This distinguishes the normalization/geometric pruning (NGP) algorithm from previously-published bounding-box algorithms.

3.4 Dynamic Geometric Pruning

The heart of the NGP algorithm is to prune the CSG tree at certain key points during normalization, thereby restricting the growth of the normalized tree before it ever gets large. In order to recognize null subtrees as early as possible, we add one more set equivalence to the normalization algorithm:

$$6.5 \qquad X - Y \cap Z = X \cap Z - Y$$

This transformation forces intersections to the left in products. Since intersection can reduce the size of bounding boxes, but subtraction and union cannot, this allows subtracted subtrees to be pruned that might not be pruned otherwise. It also allows null subtrees to be detected as early as possible, rather than having the normalization algorithm manipulate them many times before pruning.

We compute new bounding boxes after applying each set equivalence and after normalizing a left or right child. Pruning depends only on local information (at most the bounding boxes of two nodes), and is therefore very fast.

3.5 Difference Pruning

The rendering algorithm described in Section 2 requires that each primitive in a product be pruned by the remaining primitives in the product. Consider the example in Figure 7 in which *B* and *C* interact with *A*, but not with each other. The three terms that need to be rendered into the temporary buffer are:

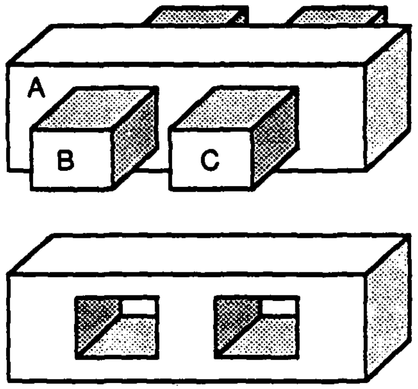

Figure 7: Difference Pruning

1. Front(A) - B - C
2. Back(B) ∩ A - C
3. Back(C) ∩ A - B

In term 2, primitive C cannot pare anything from primitive B, so primitive C can be eliminated from the term. Similarly, primitive B can be eliminated from term 3. This produces the simpler expression:

1. Front(A) - B - C
2. Back(B) ∩ A
3. Back(C) ∩ A

We check for such occurrences by comparing the bounding boxes of each primitive in a product with the bounding boxes of each of the others. We store the results of these comparisons in a matrix, whose rows and columns are indexed by the order of primitives in the product. An entry of 1 at position (i, j) means that the bounding boxes of primitives i and j intersect, and that we must pare primitive i with primitive j as usual. An entry of 0 means that a subtraction primitive j can be skipped when rendering primitive i. Since paring one primitive with another requires comparison with both front and back faces of a primitive, difference pruning results in a significant speedup for product terms with disjoint subtraction primitives.

We have shown that geometric pruning and difference pruning can result in significant savings when rendering. Section 5 analyzes and presents data on the effectiveness of these pruning algorithms.

4 Rendering Non-Convex Primitives

We say that a solid is *k-convex* if a ray intersecting the solid can enter and exit it at most k times. A 1-convex solid is convex in the usual sense and 2-convex solids include primitives such as the torus and the helix primitive (i.e. one turn of a coil spring).

To be able to render a k-convex primitive, it boundary surface must be divided into entirely front and back-facing subsurfaces. Planar polygons satisfy this condition. We assume primitives are tiled with polygons in the algorithm below.

The *front* and *back* routines now must send the polygons of a k-convex primitive to the frame buffer k times. In each pass we retain a portion of the primitive's surface, perform inside/outside classification with respect to the other primitives in the product term, and then composite what remains into the final image buffer. We use a small buffer (of size $\log_2 k$) called *count* in the frame buffer to ensure that all potentially-visible points on the primitive are captured in at least one of the passes. If a surface point is captured more than once no harm will be done. The following two algorithms are called within loops inside *front* and *back*, respectively, and perform one front or back-surface pass:

```
procedure front_pass(P: k-convex primitive, n: integer);
    { called by front $k$ times for uncomplemented $k$-convex primitives }
    begin
        .$count = 0$;
        for each front-facing polygon of P do
            for each pixel in polygon do
                begin
                    $count = count + 1$;
                    if $count = n$
                        store polygon $z$ in $ztemp$ and store polygon color in $ctemp$
                end;
    end front_pass;

procedure back_pass(P: k-convex primitive, n: integer);
    { same as front_pass, but for complemented $k$-convex primitives }
```

A 1-bit parity flag at each pixel is used to perform inside/outside classification with respect to a k-convex primitive.

```
procedure in_out_classify(P: k-convex primitive);
    { toggle $parity$ each time a surface is encountered with $z < ztemp$ }
    begin
        $parity = 0$;
        for each polygon of P do
            for each pixel in polygon do
                if $z$ of polygon $< ztemp$ then toggle $parity$
    end in_out_classify;

procedure intersect(P: k-convex primitive);
    { pare a term using an uncomplemented $k$-convex primitive }
    begin
        in_out_classify(P);
        for each pixel do
            if $parity = 0$ then $ztemp = zfar$;
    end intersect;

procedure subtract(P: k-convex primitive);
    { pare a term using a complemented $k$-convex primitive }
    begin
        in_out_classify(P);
        for each pixel do
            if $parity = 1$ then $ztemp = zfar$;
    end subtract;
```

The procedures *front_pass* and *back_pass* are called k times from the procedures *front* and *back* of Section 2.3. *Intersect* and *subtract* replace the corresponding routines in that section. These routines allow 1-convex and k-convex primitives to be rendered simultaneously without sacrificing any of the speed of the 1-convex algorithms.

5 Performance and Implementation Results

In this section we compare the performance of the NGP algorithm with competitive algorithms and present statistics taken from our modeling system. We restrict our analysis to depth-buffering algorithms for pixel-parallel architectures, noting that algorithms and architectures that exploit object parallelism, such as Kedem and Ellis' Ray Casting Machine [KE84], also have the potential for real-time CSG rendering.

5.1 Time Complexity

Two classes of depth-buffering algorithms for pixel-parallel architectures have been proposed: those which render from the original CSG tree (Rossignac/Requicha [RR86] and Jansen [Jan86]) and those which restructure the CSG tree before rendering (the NGP algorithm).

A convenient metric for comparing depth-buffering algorithms is the number of times a front or back surface of a primitive needs to be sent to the frame buffer for a particular image. In a pixel-parallel machine, the various pixel-oriented "bookkeeping" operations are very fast. We will ignore them in our analysis.

Both the Rossignac/Requicha and Jansen algorithms require sending either the front or back surface of each primitive, then performing an in/out classification [Til80] with respect to all the other primitives in the tree. For a tree with n primitives, this results in a time complexity of $n(2(n-1))$ or $O(n^2)$. This is confirmed in [RR86, page 37].

In calculating the time complexity of the NGP algorithm we will ignore the time required to normalize and prune the tree, since we have found this to be negligible for all of the objects we have rendered. Assuming that the normalized tree has j products, each of length k, the time complexity for the NGP algorithm is $jk(2(k-1)) = 2jk(k-1)$. The relationship between j and k varies depending on an object's CSG expression and the primitive geometries. Thus the algorithm's overall time complexity depends on the structure of the tree as well as the number of primitives. Pathological objects will always exist for which the NGP algorithm's time complexity explodes. Nevertheless, its performance on actual objects seems to be very good. From the limited data we have taken, it appears to perform somewhere between $O(n)$ and $O(n^2)$.

5.2 Actual Performance on Sample Data sets

We have obtained CSG data sets from a number of sources[1] and local students have used our interactive modeler to design a number of their own. Figure 8 gives statistics and rendering times for 11 objects ranging from 11 to 180 primitives. The table illustrates how small the average length of each product term (k) is after pruning (between 2 and 8). With k-values so small and relatively independent of the number of primitives the NGP algorithm's time complexity approaches $O(n)$.

5.3 Implementation Results

We have incorporated these algorithms into an interactive CSG modeler. The modeler displays a smooth-shaded, hidden-surface-removed image of the objects in the Pixel-planes 4 monitor. The user interface runs on the host DEC VaxStation II/GPX running Ultrix 2.2 using X-windows. It allows one to manipulate both the geometric descriptions of primitives and the structure of the CSG tree.

Figure 9 shows the Pixel-Planes 4 monitor and the user interface screen of our modeling system. The user can modify the CSG tree by dragging icons and can alter the sizes and positions of primitives and subtrees using joysticks [DM87]. Changes are displayed on the Pixel-planes 4 monitor in fractions of a second for objects between 20 and 50 primitives. Figure 10

[1]Note to readers of draft 1/11/88: We have recently received a number of large CSG data sets from the University of Rochester and IBM. We have not yet had time to convert them to our format, but will include data and images from them in future drafts of this paper and on the accompanying videotape.

Object Description			Number of primitives				Primitive faces sent to frame buffer		Average product length (k)	Rendering time
No	Part Name	Source	user tree	after pruning	norm tree	after pruning	w/out diff pruning	with diff pruning		
1.	Tube	[OKM84]	11	11	15	15	75	75	3.5	0.185 sec
2.	Cut Tube	[OKM84]	12	12	20	20	140	136	4.4	0.235
3.	MBB	[OKM84]	23	23	30	30	150	150	3.5	0.138
4.	Tie Rod	BRL[1]	19	19	19	19	127	91	3.4	0.142
5.	Phone	Randy Brown	32	32	62	31	852	456	8.4	0.110
6.	Lamp	Brice Tebbs	15	15	16	16	32	32	2.0	0.225
7.	Die	Greg Turk	44	44	44	44	380	252	3.9	4.020
8.	Meal	David Banks	43	43	43	43	711	456	6.3	3.430
9.	GEB[2]	Steven Molnar	29	28	297	184	2252	2036	7.2	0.711
10.	Truck	BRL[1]	180	180	180	180	352	352	2.0	2.770
11.	Tie Rod - MBB(above)		42	42	69120	96	1152	928	5.8	0.592

[1] US Army Ballistic Research Laboratory
[2] Inspired by cover of Godel, Escher, Bach, by Douglas Hofstadter

Figure 8: Rendering times and statistics

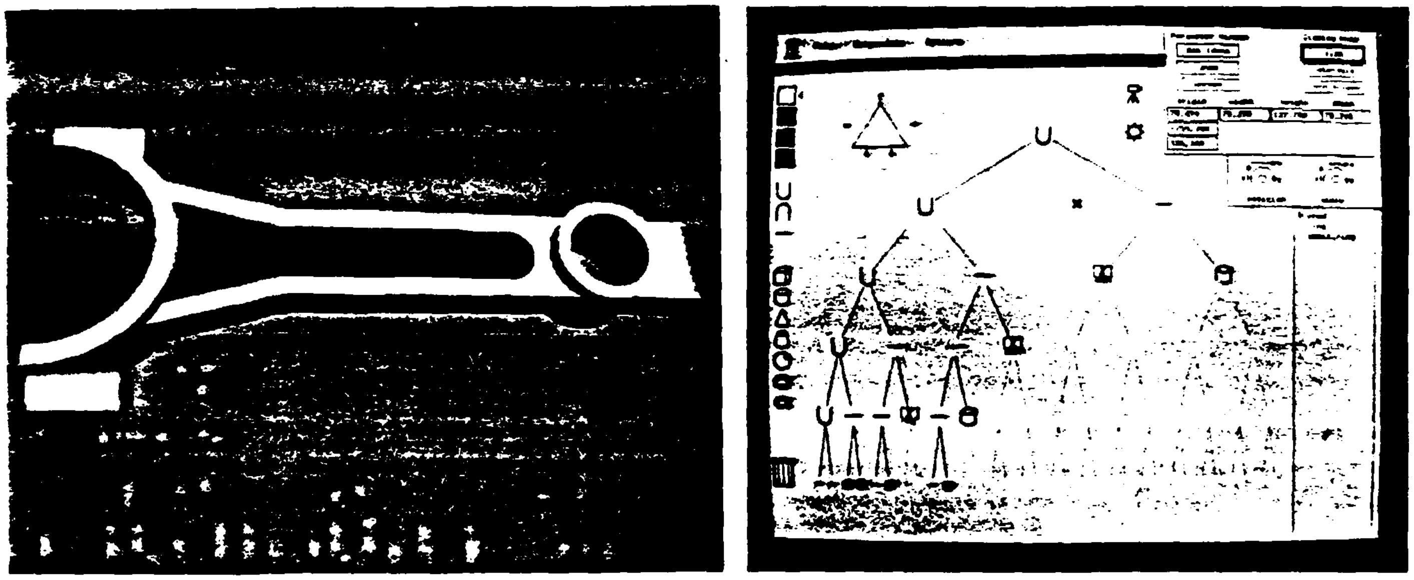

Figure 9: Pixel-Planes 4 monitor and modeler interface screen

shows images and rendering times for a number of the objects described in the table of Figure 8.

Pixel-planes 4's front-end processor is a single fast-floating point processor based on the Weitek floating point chip set. We have found this to be the bottleneck in our system. The majority of the CSG objects we have rendered use only approximately ten percent of Pixel-planes 4's frame buffer cycles. This makes us believe that the NGP algorithm may perform better on systems with more front-end processing power, in particular, high-performance graphics workstations with large-grain parallelism among front-end processors.

5.4 Future Extensions

We intend to increase the speed of our renderer in two ways: 1) We can increase the power of our front-end processor. The NGP algorithm adapts well to the large-grain MIMD front-end paradigm. Jansen in [JS87] describes an algorithm for a multi-processor system that divides the frame buffer into multiple regions, allocating a processor per region. Each processor maintains its own version of the CSG tree and prunes it using its particular viewing frustum as a bounding box. We intend to implement this algorithm on Pixel-planes 5 (currently under development). 2) Also we can use the quadratic expression evaluator of Pixel-planes 5 to render primitives with quadratic surfaces. This will allow us to render primitives with curved surfaces rapidly.

We also plan to enhance the pruning algorithm and make it less view-dependent using tighter bounding boxes. We intend to replace bounding boxes with Kay and Kajiya's bounding regions [KK86].

6 Conclusion

We have presented a set of algorithms for efficiently rendering CSG-described objects directly into a frame buffer without first converting to a boundary representation. Unlike other depth-buffering CSG algorithms, the NGP algorithm requires only a constant number of bits per pixel (two image buffers plus three flag bits) to render objects with convex primitives. It also can take advantage of pixel-parallelism available in the new generation of high-performance

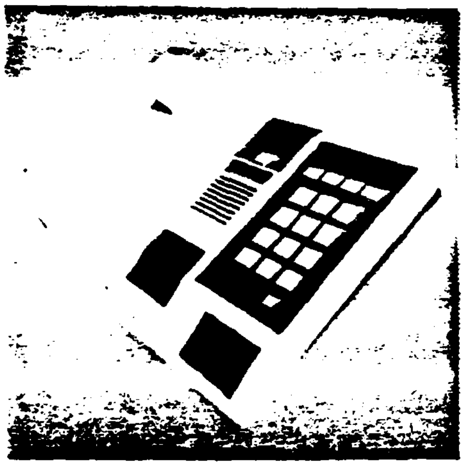
Fig 10a: Phone (0.110 sec)

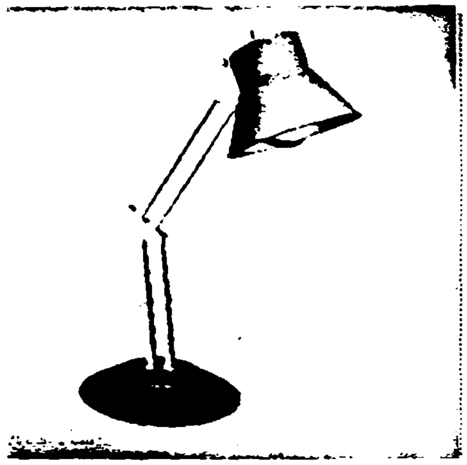
Fig 10b: Lamp (0.225 sec)

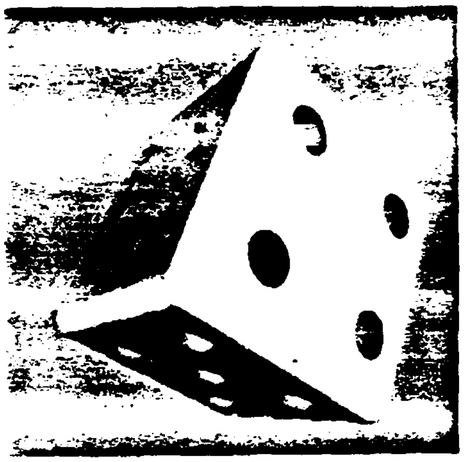
Fig 10c: Die (4.020 sec)

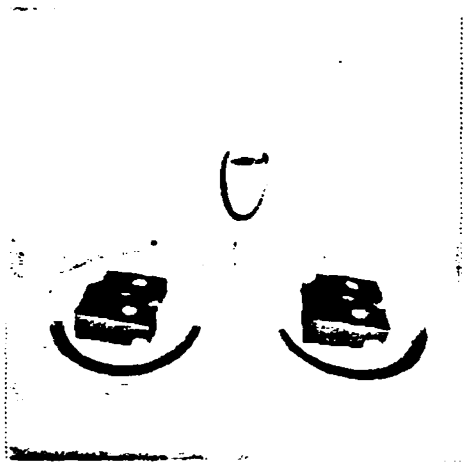
Fig 10d: Meal (3.430 sec)

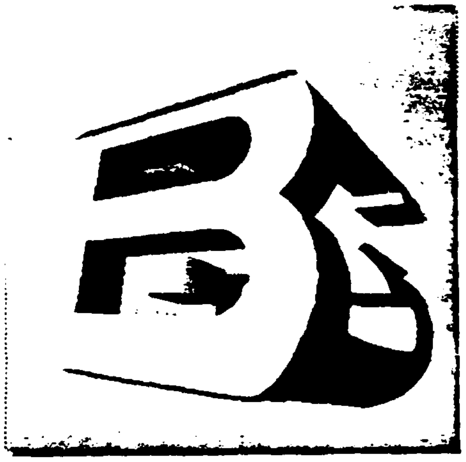
Fig 10e: GEB (0.711 sec)

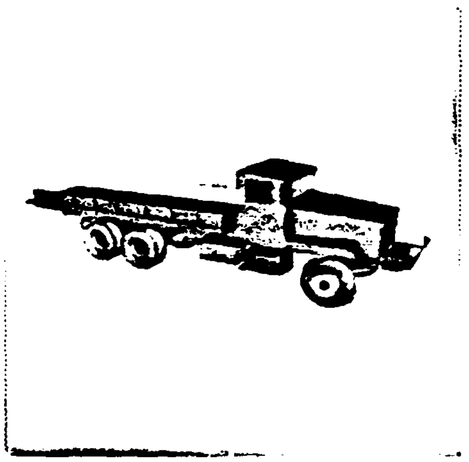
Fig 10f: Truck (2.770 sec)

Figure 10: Images generated on Pixel-planes 4 solid modeler

graphics workstations such as the ATT Pixel Machine and the forthcoming Stellar graphics workstation. Although the NGP algorithm has a worst-case time complexity much worse than other algorithms' $O(n^2)$, on all of the trees we have encountered, its time complexity compares favorably. We have incorporated the NGP algorithm into an interactive CSG modeling system that has proven very successful. It renders shaded images of substantial objects in a fractions of a second. This "what you see is what you get" (wysiwyg) approach to solid modeling has attracted students outside of the Pixel-planes group, several of whom have designed substantial CSG objects with very little instruction.

References

[Ath83] Peter R. Atherton. A scan-line hidden surface removal procedure for constructive solid geometry. *Computer Graphics (Proceedings of SIGGRAPH '83)*, 17(3):73–82, 1983.

[BG82] John W. Boyse and Jack E. Gilchrist. Gmsolid: interactive modeling for design and analysis of solids. *IEEE Computer Graphics and Applications*, 2(2):27–40, March 1982.

[BMPA87] Kadi Bouatouch, M. O. Madani, Thierry Priol, and Bruno Arnaldi. A new algorithm of space tracing using a csg model. *Proceedings Eurographics '87, Elseviers Science Publ.*, 65–77, 1987.

[Bro82] C. M. Brown. Padl-2: a technical summary. *IEEE Computer Graphics and Applications*, 2(2):69–84, March 1982.

[Cha86] Edward Chang. Brown animation generation system (bags). June 1986. Computer Graphics Group, Department of Computer Science, Brown University.

[DM87] Clare Durand and Steven Molnar. Csg user's manual. February 1987. Computer Science Dept., University of North Carolina at Chapel Hill.

[For86] A. R. Forrest. User interfaces for three-dimensional geometric modeling. *Proceedings of Chapel Hill Workshop on Interactive 3D Graphics*, 6, 1986.

[GHF86] Jack Goldfeather, Jeff P.M. Hultquist, and Henry Fuchs. Fast constructive solid geometry display in the pixel-powers graphics system. *Computer Graphics (Proceedings of SIGGRAPH '86)*, 20(4):107–116, 1986.

[GM88] Jack Goldfeather and Steven Molnar. Csg tree restructuring. January 1988. Technical Report (forthcoming): Computer Science Dept., University of North Carolina at Chapel Hill.

[Jan86] F. W. Jansen. A pixel-parallel hidden surface algorithm for constructive solid geometry. *Proceedings Eurographics '86, Elseviers Science Publ.*, 29–40, 1986.

[JS87] F. W. Jansen and R. J. Sutherland. Display of solid models with a multi-processor system. *Proceedings Eurographics '87, Elseviers Science Publ.*, 377–387, 1987.

[KE84] Gershon Kedem and J. L. Ellis. Computer structures for curve-solid classification in geometric modeling. 1984. Technical Report TR84-37, Microelectronic Center of North Carolina, Research Triangle Park, N.C.

[KK86] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. *Computer Graphics (Proceedings of SIGGRAPH '86)*, 20(4):269–278, 1986.

[Koh78] Zvi Kohavi. *Switching and Finite Automata Theory*. Tata McGraw-Hill Publishing Co. Ltd., New Delhi, 1978.

[OKM84] N. Okino, Y. Kakazu, and M. Morimoto. Extended depth buffer algorithms for hidden surface visualization. *IEEE Computer Graphics and Applications*, 4(5):79–88, May 1984.

[RR86] J. R. Rossignac and A. A. G. Requicha. Depth buffering display techniques for constructive solid geometry. *IEEE Computer Graphics and Applications*, 6(9):29–39, September 1986.

[Til80] R. B. Tilove. Set membership classification: a unified approach to geometric intersection problems. *IEEE Transactions on Computers*, C-29(10):874–883, October 1980.

[TN87] William C. Thibault and Bruce F. Naylor. Set operations on polyhedra using binary space partitioning trees. *Computer Graphics (Proceedings of SIGGRAPH '87)*, 21(4):153–162, 1987.

[TSUC86] Hiroshi Toriya, Toshiaki Satoh, Kenji Ueda, and Hiroaki Chiyokura. Undo and redo operations for solid modeling. *IEEE Computer Graphics and Applications*, 6(4):35–42, April 1986.